CORPORACIÓN UNIVERSITARIA AUTÓNOMA DEL CAUCA

PROGRAMACION II

ENTREGABLE EJERCICIOS CLASE 11

PRESENTADO POR:

SALOMÓN MONTILLA LUNA

PRESENTADO A:

CRISTHIAN ALEJANDRO CAÑAR MUÑOZ

POPAYÁN, CAUCA

2025

DESARROLLO

```java
    static void metodoRecursivo() {  2 usages  new *
        metodoRecursivo();
    }
    public static void main(String[] args) {  Salomon *
        // Manejo de una Excepción (Checked Exception)
        try {
            FileReader file = new FileReader( fileName: "archivo_no_existente.txt"); // Generará IOException
        } catch (IOException e) {
            System.out.println("¡Excepción capturada! Archivo no encontrado: " + e);
        }
        // Manejo de una Excepción (Unchecked Exception)
        try {
            int resultado = 10 / 0; // División por cero genera ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("¡Excepción capturada! División por cero: " + e);
        }
        // Generar un Error (OutOfMemoryError)
        try {
            List<int[]> lista = new ArrayList<>();
            while (true) {
                lista.add(new int[1000000]); // Consumirá toda la memoria heap
            }
        } catch (OutOfMemoryError e) {
            System.out.println("¡Error crítico! Memoria insuficiente: " + e);
        }
        // Generar un Error (StackOverflowError)
        try {
            metodoRecursivo();
        } catch (StackOverflowError e) {
            System.out.println("¡Error crítico! Desbordamiento de pila: " + e);
        }
    }
}
```

```
¡Excepción capturada! Archivo no encontrado: java.io.FileNotFoundException: archivo_no_existente.txt (El sistema no puede encontrar el archivo especificado)
¡Excepción capturada! División por cero: java.lang.ArithmeticException: / by zero
¡Error crítico! Memoria insuficiente: java.lang.OutOfMemoryError: Java heap space
¡Error crítico! Desbordamiento de pila: java.lang.StackOverflowError
```

1. Explicación teórica:

a. ¿Cuál es la diferencia entre Exception y Error en Java?

La diferencia principal es que las *Exception* son situaciones que pueden pasar en el programa y que normalmente podemos controlar, como cuando intentamos dividir por cero o abrir un archivo que no existe. En cambio, los *Error* son problemas más graves que vienen del sistema o del entorno de Java, como cuando se acaba la memoria. Esos errores casi siempre son cosas que no podemos (ni deberíamos) manejar desde el código.

b. **¿Por qué IOException es una Checked Exception y ArithmeticException una Unchecked Exception?**

IOException es *checked* porque Java quiere que el programador la maneje sí o sí con try-catch o lanzándola con throws para no interrumpir el flujo del programa, ya que ocurre por cosas externas como leer archivos inexistentes, en cambio, ArithmeticException es *unchecked* porque es un error lógico del código, como dividir por cero, y Java espera que el programador tenga cuidado con eso sin obligarlo a usar try-catch. (Martin, 2024)

c. **¿Es recomendable manejar errores (Error) con try-catch? Justifica tu respuesta.**

No es recomendable porque los *Error* son cosas muy graves como que se quedó sin memoria o falló Java. Tratar de atraparlos con try-catch no sirve mucho porque ya el programa está funcionando incorrectamente. Lo mejor es evitar que pasen y no atraparlos.

2. **Análisis de ejecución:**

a. **¿En qué orden se generaron los errores y excepciones?**

Primero se genera la IOException, porque el código intenta abrir un archivo que no existe. Luego se lanza la ArithmeticException por dividir entre cero. Después, se genera un OutOfMemoryError porque el programa está creando arreglos enormes en un ciclo infinito y se queda sin memoria. Finalmente, se lanza el StackOverflowError cuando el método metodoRecursivo() se llama a sí mismo infinitamente sin condición para detenerse.

Entonces, el orden es:

1. IOException
2. ArithmeticException
3. OutOfMemoryError

4. StackOverflowError

b. ¿Cómo podrías evitar el OutOfMemoryError en este código?

Una forma sería no usar un ciclo infinito que crea tantos arreglos grandes, porque eso hace que el programa siga consumiendo memoria sin parar hasta que se llena toda la memoria disponible. En vez de eso, se puede poner una condición que limite la cantidad de arreglos creados, o simplemente evitar crear arreglos tan grandes. Así se evita que se acabe la memoria y se lance el *OutOfMemoryError*.

c. ¿Cómo evitarías el StackOverflowError sin modificar el try-catch?

Para evitarlo, hay que agregarle un caso base que detenga la recursión en cierto punto. Por ejemplo, usar un parámetro como un contador que se detenga al llegar a un número. También se puede cambiar la lógica para usar un bucle while o for si no es necesario que sea recursivo.

```java
public class Solucion {  new *
    public static void main(String[] args) {  new *
        // Manejo de una Excepción evitando la IOException
        try {
            File archivo = new File( pathname: "archivo_no_existente.txt");
            if (archivo.exists()) {
                FileReader file = new FileReader(archivo);
            } else {
                System.out.println("El archivo no existe.");
            }
        } catch (IOException e) {
            System.out.println("¡Excepción capturada! Archivo no encontrado: " + e);
        }

        // Manejo de una Excepción evitando la ArithmeticException
        try {
            //int divisor = 0;
            int divisor = 2; // Cambiamos el divisor a un valor diferente de cero
            if (divisor != 0) {
                int resultado = 10 / divisor;
                System.out.println("Manejada -> Resultado: " + resultado);
            } else {
                System.out.println("No se puede dividir por cero.");
            }
        } catch (ArithmeticException e) {
            System.out.println("¡Excepción capturada! División por cero: " + e);
        }

        // Manejo de Error (OutOfMemoryError) - evitando el error sin usar try-catch
        List<int[]> lista = new ArrayList<>();
        for (int i = 0; i < 10; i++) { // Limita la cantidad de arreglos creados
            lista.add(new int[1000000]);
        }

        // Manejo de Error (StackOverflowError) - evitando el error sin usar try-catch
        metodoRecursivoSeguro( contador: 0); // Agregamos condición de parada
    }

    public static void metodoRecursivoSeguro(int contador) {  2 usages  new *
        if (contador >= 1000) return; // condición de parada para evitar el StackOverflow
        metodoRecursivoSeguro( contador: contador + 1);
    }
}
```

```
El archivo no existe.
Manejada -> Resultado: 5
```

BIBLIOGRAFÍA

Martin, E. (8 de Enero de 2024). *Checked and Unchecked Exceptions in Java*. Obtenido de https://www.baeldung.com/java-checked-unchecked-exceptions